Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

MAS

Modelling, Analysis and Simulation

Modelling, Analysis and Simulation

Analysis and optimization of an algorithm for discrete tomography

K.J. Batenburg

REPORT MAS-E0327 DECEMBER 22, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3703

Analysis and optimization of an algorithm for discrete tomography

ABSTRACT
Binary tomography is concerned with recovering binary images from a finite number of discretely sampled projections. Hajdu and Tijdeman outlined an algorithm for this type of problem. In this paper we analyze the algorithm and present several ways of improving the time complexity. We also give the results of experiments with an optimized version which is much faster than the original implementation, up to a factor of 50 or more (depending on the problem instance).

2000 Mathematics Subject Classification: 65F25
Keywords and Phrases: discrete tomography, QR decomposition
Note: This work was carried out under project MAS 2.2.

Analysis and optimization of an algorithm for discrete tomography

K.J. Batenburg
Centrum voor Wiskunde en Informatica

December 22, 2003

Abstract

Binary tomography is concerned with recovering binary images from a finite number of discretely sampled projections. Hajdu and Tijdeman outlined an algorithm for this type of problem in [6]. In this paper we analyze the algorithm and present several ways of improving the time complexity. We also give the results of experiments with an optimized version which is much faster than the original implementation, up to a factor of 50 or more (depending on the problem instance).

1 Introduction

Binary tomography is concerned with recovering binary images from a finite number of discretely sampled projections. The main problem is to reconstruct a function $f : A \to \{0, 1\}$ where A is a finite subset of $\mathbb{Z}^l$ ($l \geq 2$), if the sums of the function values along all the lines in a finite number of directions are given. Typically, line sums are only available in a few directions. The corresponding system of equations is very underdetermined and may have a large class of solutions. The structure of this solution class was studied by Hajdu and Tijdeman in [5]. They showed that the solution set of 0-1 solutions is precisely the set of shortest vector solutions in the set of $\mathbb{Z}$-solutions. By $\mathbb{Z}$-solutions we mean functions $A \to \mathbb{Z}$ with the given line sums. It is also shown in [5] that the $\mathbb{Z}$-solutions form a multidimensional grid on a linear manifold (containing the $\mathbb{R}$-solutions) in a real vectorspace. The dimension of this vectorspace is the number of elements in A.
The two results from [5] mentioned above form the basis for an algorithm for solving the binary tomography problem, proposed in [6]. An important operation in this algorithm is the Projection operation. This operation involves computing the orthogonal projection of the origin onto a linear manifold. Because the operation is executed many times and it is very time-consuming on larger problem instances, we will investigate a method for reducing the time complexity of the operation.
This paper is an extended version of [8], in which new results have been incorporated, in particular in Section 4. Its main purpose is to describe several computational aspects of the Projection operation and to present a method for reducing the time complexity in some cases. We also describe several other ways of improving the performance of the algorithm. Our practical results show a great improvement in runtime over the original implementation from [6].

2 Notation and concepts

The binary tomography problem that we consider in this paper can be stated as follows:

Problem 2.1 *Let k, m, n be integers greater than 1. Let*

$$A = \{(i, j) \in \mathbb{Z}^2 : 0 \leq i < m, 0 \leq j < n\}$$

and $f : A \to \{0, 1\}$. Let $D = \{(a_d, b_d)\}_{d=1}^k$ be a set of pairs of coprime integers. Suppose f is unknown, but all the line sums $\sum_{a_d j = b_d i + t} f(i, j)$ (taken over $(i, j) \in A$) are given for $d = 1, \ldots, k$ and $t \in \mathbb{Z}$. Construct a function $g : A \to \{0, 1\}$ such that

$$\sum_{a_d j = b_d i + t} f(i, j) = \sum_{a_d j = b_d i + t} g(i, j) \quad \text{for } d = 1, \ldots, k \quad \text{and} \quad t \in \mathbb{Z}.$$

We call all pairs (i, j) such that $a_d j = b_d i + t$ for any fixed t a *line* and the corresponding sum a *line sum*. For the theoretical treatment we will restrict ourselves to the case where A is a two-dimensional array, but generalization of the presented material to the case where A is an l-dimensional array with $l > 2$ is straightforward. In fact, we will show in Section 5.3 that the presented algorithm can be used for the case $l > 2$ without any major modification. We will use the definitions of A and D from Problem 2.1 throughout the rest of this paper. When trying to solve Problem 2.1 it is sometimes useful to relax the constraint that the image of the functions f and g must be binary. Therefore we will also use a modified version of Problem 2.1 where we consider maps from A to R for any commutative ring R, instead of the set $\{0, 1\}$. In particular, the cases $R = \mathbb{Z}$ and $R = \mathbb{R}$ are both relevant for this study. We will denote these cases with (2.1a) and (2.1b) respectively.
A matrix $M \in \mathbb{R}^{m \times n}$ corresponds directly to a function $f : A \to \mathbb{R}$:

$$f(i, j) = M_{i+1,j+1} \quad \text{for} \quad 0 \leq i < m, 0 \leq j < n.$$

We will call M the *matrix representation* of f.
Another representation that we will use is the *vector representation*. In order to write the linesum-constraints on a matrix $M \in \mathbb{R}^{m \times n}$ as a system of linear equations, having the elements of M as its variables, we regard the matrix M as a vector. Let $v \in \mathbb{R}^{mn}$. We say that M and v *correspond* if and only if

$$M_{ij} = v_{(i-1)n+j} \quad \text{for} \quad 1 \leq i \leq m, 1 \leq j \leq n.$$

The vector representation defines an order on the elements of M. From this point on, we will use the term *entry* k to denote the kth entry in the vector representation. Throughout this paper, we will use the function, matrix and vector representations interchangeably.
Let s be the number of given line sums. We define the $s \times mn$-matrix B:

$$B_{t,k} = \begin{cases} 1 & \text{if line } t \text{ contains entry } k \\ 0 & \text{otherwise} \end{cases} \qquad \text{for} \quad t = 1, \ldots, s;\ k = 1, \ldots, mn.$$

We call B the *line sum-matrix*. The constraints on the line sums of a solution M to Problem 2.1b can now be formulated as a system of real linear equations that the corresponding vector-representation v must satisfy:

$$Bv = b \tag{1}$$

We define the l_2-norm $\|v\|_2 = \sqrt{\sum_{k=1}^{mn} v_k^2}$ on the vector-representation v.
In this study our starting point will be the algorithm that is presented in [6]. We summarize the results from [5] on which the algorithm is based. Let R be a commutative ring. We can regard the set of functions $F = \{f : A \to R\}$ as a vector space over R. The set of functions that have zero linesums along all directions of D corresponds to a linear subspace, F_z, of F.

Theorem 2.2 *Let $m, n \in \mathbb{N}$ and put $M = \sum_{d=1}^{k} a_d$, $N = \sum_{d=1}^{k} |b_d|$. Put $m' = m - 1 - M, n' = n - 1 - N$. Let R be an integral domain such that $R[x, y]$*

is a unique factorization domain. Then for any nontrivial set of directions D there exist functions

$$m_{uv} : A \to R \qquad u = 0, \ldots, m'; v = 0, \ldots, n'$$

such that

$$F_z = \text{span}\{m_{uv} : \quad u = 0, \ldots, m'; v = 0, \ldots, n'\}$$

and any function $g : A \to R$ with zero line sums along the lines corresponding to D can be uniquely written in the form

$$g = \sum_{u=0}^{m'} \sum_{v=0}^{n'} c_{uv} m_{uv}.$$

A proof of Theorem 2.2 is given in [5], where an explicit way of constructing the functions m_{uv} is presented. According to this theorem, the functions m_{uv} form a basis of F_z. We see that if g, h are both solutions to Problem 2.1, the difference $g - h$ can be written as a linear combination of the functions m_{uv}, since it has zero linesums in all given directions.
An illustration of the matrices m_{uv} for the case when all linesums in the horizontal, vertical, diagonal and antidiagonal directions are given is presented in [6]. The set of matrices m_{uv} consists of the translates of the nonzero part of a single matrix. This nonzero part is independent of the array-size parameters m, n. For large arrays, the matrices m_{uv} are only nonzero on a small, local group of entries. Because of the characteristic shape of the set of nonzero entries for the sample case, the authors of [6] denote the matrices m_{uv} by the term *mills*, even in the general case where the set of directions D is variable. We will also use this term.
The following problem connects Problem 2.1a (the integer case) to Problem 2.1 (the binary case):

Problem 2.3 *Construct a function $g : A \to \mathbb{Z}$ such that g is a solution to Problem 2.1a and*

$$\sum_{(i,j) \in A} g(i,j)^2 \quad \textit{is minimal.}$$

Remark 2.4 Problem 2.3 is a generalization of Problem 2.1, because for any $f : A \to \{0, 1\}$ that is a solution to Problem 2.1a:

$$\sum_{(i,j) \in A} f(i,j)^2 = \sum_{(i,j) \in A} f(i,j) = \sum_{(i,j) \in A} g(i,j) \leq \sum_{(i,j) \in A} g(i,j)^2$$

with equality if and only if $\text{Im}(g) \subseteq \{0, 1\}$. Therefore an algorithm that is capable of solving Problem 2.3 is also capable of solving Problem 2.1.

3 Description of the algorithm

We will now describe the most important features of the algorithm from [6]. The original paper offers a much greater level of detail than the material presented here. For a concise description we refer to Section 3 and 4 of the original paper. From this point on we will only use the vector representation (as opposed to the function or matrix representation) for intermediate solutions. We will use the same notation as introduced in Problem 2.1. At any point in the algorithm we denote the current solution by $\tilde{x}$.
The algorithm tries to find a solution to Problem 2.3. The first step is to construct a set of functions

$$m_{uv} : A \to R \qquad\qquad u = 0, \ldots, m', \quad v = 0, \ldots, n'$$

that have zero line sums for all directions in D as described in Theorem 2.2, for the case $R = \mathbb{R}$. These functions can be stored in the vector representation.
The next step is to compute a solution of the real Problem 2.1b. This comes down to solving the system of real linear equations that corresponds to the given line sum constraints. Because this system is usually underdetermined (the number of variables is much greater than the number of line sums) many solutions may exist.
According to Remark 2.4, the set of solutions of the binary Problem 2.1 is exactly the set of solutions of the integer Problem 2.1a that have minimal length with respect to the l_2-norm. Therefore we use the solution x^* of the real Problem 2.1b that has minimal l_2-norm as a starting value for $\tilde{x}$. Because $\|y\|_2$ is the same for all binary solutions y, it follows from the Pythagorean formula that all binary solutions lie on a hypersphere centered in x^* in the real manifold $W = \{x \in \mathbb{R}^n : Bx = b\}$, where B is the line sum-matrix.
Next, the algorithm enters the main loop. The general idea is that we can modify the current real solution $\tilde{x}$ by adding linear combinations of the mills m_{uv} without changing the line sums. The algorithm tries to add these linear combinations in such a way that the entries of $\tilde{x}$ become integer values, preferably 0 or 1. In each iteration, one of the functions m_{uv} is *fixed*. This means that it will not be used to modify the solution in future iterations.
We say that a mill m_{uv} *overlaps* an entry $\tilde{x}_i$ if the corresponding entry of m_{uv} is nonzero. When all mills that overlap $\tilde{x}_i$ have been fixed, the value of this entry cannot be modified anymore. This influences the choice which mill to fix in an iteration. In each iteration, a *border entry* is chosen. The set of border entries consists of all entries that have only one non-fixed overlapping mill. From this set an entry $\tilde{x}_{i^*}$ is chosen such that $|\tilde{x}_{i^*} - 1/2|$ is maximal. Because $|\tilde{x}_{i^*} - 1/2|$ is maximal, we can usually predict which of the values 0 and 1 entry i^* should have in the final solution. By adding a real multiple of the overlapping mill, the entry is given this value. After the entry has been given its final value, the overlapping mill is fixed. We then call the entry $\tilde{x}_{i^*}$ *fixed* as well.
The property that we can add linear combinations of the mills m_{uv} to a solution $\tilde{x}$ without violating the linesum constraints is also used in the process of *local smoothening*. This operation pulls the entries of $\tilde{x}$ towards 0 if they are negative

and towards 1 if they are greater than 1.
The operations that involve adding a linear combination of mills to the current solution all have a local effect, because every mill is nonzero for a small, local set of entries. In order to smoothen the solution globally, the *Projection operation* is used. The set *locallyfixed* is formed, which is the union of the set of fixed entries and the set of entries $\tilde{x}_i$ that are not yet fixed for which $|\tilde{x}_i - 1/2| \geq p_3$ where p_3 is a parameter of the algorithm. A natural choice for this parameter is $p_3 = 0.5$. Next, all entries in the set *locallyfixed* are temporarily fixed at binary values:

$$\tilde{x}_i = \begin{cases} 1 & \text{if } x_i \geq 1/2 \\ 0 & \text{if } x_i < 1/2 \end{cases} \qquad \text{for} \quad i \in \text{locallyfixed}$$

The system $Bx = b$ of linesum equations now becomes

$$Bx = b \quad \text{and} \quad x_i = \tilde{x}_i \quad \text{for all } i \in \text{locallyfixed}. \tag{2}$$

The solution set of this equation is a sub-manifold of the manifold
$W = \{x \in \mathbb{R}^n : Bx = b\}$. Similar to the computation of the start solution, we now compute the shortest vector in the solution manifold of (2). We repeat the projection process until either equation (2) no longer has a solution or a stop criterion is satisfied, indicating that all entries of the current solution are close to the range $[0, 1]$. The last valid solution that is found before the projection procedure finishes is used as the new current solution $\tilde{x}$. We remark that although a number of entries of $\tilde{x}$ may have been fixed during the Projection operation, all entries that were not fixed before the Projection operation can still be modified afterwards, because there are still overlapping mills that are not fixed yet. For the details concerning the Projection operation we refer to Section 4 of the original paper [6].
The algorithm terminates when all mills have been fixed. The resulting solution is guaranteed to be integral, but is not necessarily binary.

4 Analysis of the Projection operation

In this section we will analyze the Projection operation and discuss how to make it computationally efficient.

4.1 Computing the projection

According to Remark 2.4, each binary solution to Problem 2.1a has minimal l_2-norm among all integer solution vectors. When we search for binary solutions in the real manifold $W = \{x \in \mathbb{R}^n : Bx = b\}$, it seems reasonable to use the shortest vector in this manifold as a starting point for the algorithm.

Problem 4.1 *Compute the vector $x^* \in W$ such that*

$$\|x^*\|_2 = \min_{x \in W} \|x\|_2.$$

If we assume that B has full row rank, then the product BB^T is nonsingular and the unique solution of Problem 4.1, which is the orthogonal projection of the origin onto W (see, e.g., [2]), is given by

$$\tilde{x} = B^T(BB^T)^{-1}b.$$

One may compute $x^* = \tilde{x}$ by first solving the system

$$BB^T v = b$$

for v and then computing

$$x^* = B^T v.$$

The first system can be solved by using the Cholesky-factorization $BB^T = LL^T$, where L is lowertriangular. However, as is shown in [2], this method can lead to a serious loss of accuracy if the matrix BB^T is ill-conditioned.

A different approach to solving Problem 4.1 is to use the QR decomposition. We will call a square matrix Q *orthogonal* if $QQ^T = I$, where I is the identity matrix.

Definition 4.2 Let m, n be integers greater than 0 with $m \geq n$. Let $M \in \mathbb{R}^{m\times n}$. Suppose that M has full column rank. A *QR decomposition* of M has the form

$$M = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal, $R \in \mathbb{R}^{n\times n}$ is uppertriangular and nonsingular and 0 corresponds to a (possibly empty) block of rowvectors.

From the nonsingularity of R it is easy to see that the first n columns of Q form a basis for the column space of M. For each matrix M that has full column rank, there is a QR decomposition such that R has only positive diagonal elements (see, e.g., Section 4.1 of [7]). Efficient algorithms for computing the QR decomposition are described in Section 5.2 of [4] and Section 4.1 of [7].
We will solve Problem 4.1 by using the QR decomposition of $B^T \in \mathbb{R}^{mn\times s}$, where m, n are the dimensions of the array A and s is the number of given line sums. The matrix B^T however will certainly not have full column rank. For example, the sum of all line sums in any single direction must equal the total number of ones, so whenever there are more than two directions there is a linear dependence between the line sums. Put $r = \text{rank}(B)$. We will use a *QR decomposition with column pivoting*:

$$B^T\Pi = Q\begin{pmatrix} R & S \\ 0 & 0 \end{pmatrix}$$

where $Q \in \mathbb{R}^{mn\times mn}$ is orthogonal, $\Pi \in \mathbb{R}^{s\times s}$ is a permutation, $R \in \mathbb{R}^{r\times r}$ is uppertriangular and nonsingular. The bottom 0's correspond to (possibly empty) rectangular blocks. By applying Π we make sure that the first r columns of the matrix $B^T\Pi$ are linear independent. In this paper we will denote this

decomposition as the *extended QR decomposition*. An algorithm for computing the extended QR decomposition is described in Section 5.4.1 of [4]. The time complexity of computing the extended QR decomposition of a $k \times s$ matrix of rank r is $O(k^3+ksr)$. In this case $k = mn$ so the operation is $O((mn)^3+mnsr)$. We remark that the extended QR decomposition is not unique.
The first r columns of $B^T\Pi$ correspond to r rows of B that form a basis of the rowspace of B. When solving the inhomogeneous system $Bx = b$, these rows completely determine the solution manifold, unless the equations corresponding to any of the other rows make the system inconsistent. In the latter case the system has no solution. Once we have computed the solution to Problem 4.1, using only the first r columns of $B^T\Pi$, we can check that the system is consistent by substituting the solution into the remaining equations. We will now show how to compute the projection using the QR decomposition. We have

$$\begin{gathered} Bx = b \iff x^T B^T = b^T \iff \\ x^T(B^T\Pi) = b^T\Pi \iff x^T Q \begin{pmatrix} R & S \\ 0 & 0 \end{pmatrix} = b^T\Pi. \end{gathered}$$

Let $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = Q^T x$ where y_1 consists of the first r elements of y. Then

$$\begin{gathered} x^T Q \begin{pmatrix} R \\ 0 \end{pmatrix} = b^T\Pi \iff \begin{pmatrix} R^T & 0 \end{pmatrix} Q^T x = \Pi^T b \iff \\ \begin{pmatrix} R^T & 0 \end{pmatrix} y = \Pi^T b \iff R^T y_1 = \Pi^T b. \end{gathered}$$

From the fact that R is nonsingular, it follows that the last system has a unique solution. Because Q is orthogonal, we have $\|x\|_2 = \|y\|_2$. It follows that we obtain the unique solution x^* to Problem 4.1 by setting $y_2 = 0$:

$$x^* = Q \begin{pmatrix} y_1 \\ 0 \end{pmatrix} = Q(R^T)^{-1}\Pi^T b \tag{3}$$

When the extended QR decomposition of B^T is known, the vector x^* can be computed efficiently by first solving the system

$$R^T y_1 = \Pi^T b$$

for y_1 and then computing

$$x^* = Q \begin{pmatrix} y_1 \\ 0 \end{pmatrix}.$$

The first computation can be performed in $O(r^2)$ time by forward substitution (R^T is lower-triangular), the second computation is $O(mnr)$. We see that in this procedure for computing x^*, computing the extended QR decomposition is by far the operation with the worst time complexity.
The numerical properties of using the QR decomposition for solving Problem 4.1 are very favorable in comparison to using the Cholesky decomposition (see

[2]). Moreover, we will show in the next subsections that this decomposition can be adapted dynamically so that it can be reused many times during each run of the algorithm. We remark that our proposed method for computing the projection is not used in the original paper [6].

4.2 Updating the projection: fixing variables

In every iteration of the algorithm one of the mills is fixed. As a consequence the value of certain entries of the solution $\tilde{x}$ becomes fixed as well. The Projection operation also involves fixing the values of solution entries, though temporarily. Suppose that when the Projection operation is executed the entries in $I = \{i_1, \ldots, i_k\}$ are either already fixed or fixed temporarily at integer values. We now project the origin onto the solution manifold of the system

$$Bx = b \quad \text{and} \quad x_{i_1} = v_{i_1},\ x_{i_2} = v_{i_2},\ \ldots,\ x_{i_k} = v_{i_k} \tag{4}$$

where v_{i_t} is the fixed value of the entry i_t. Solving this system is equivalent to solving the system

$$\tilde{B}\tilde{x} = \tilde{b} \tag{5}$$

where $\tilde{B}$ is obtained by removing the columns $B_{i_1}, \ldots, B_{i_k}$ from B and setting

$$\tilde{b} = b - (\sum_{t=1}^{k} v_{i_t} B_{i_t}).$$

For each solution vector $\tilde{x}$, the corresponding vector x can be computed by assigning the values $v_{i_t} (t = 1, \ldots, k)$ to the corresponding fixed entries of x and assigning the entry values of $\tilde{x}$ to the corresponding nonfixed entries of x.
The projection of the origin onto the solution manifold of (5) can be found by the procedure that is described in Section 4.1. However, this operation is computationally very expensive. The operation may have to be executed many times if between subsequent projections only a few entries have been fixed. Suppose that, in order to compute the projection of the origin onto the solution manifold of (5), we have computed the extended QR decomposition of $\tilde{B}^T$:

$$\tilde{B}^T \tilde{\Pi} = \tilde{Q} \begin{pmatrix} \tilde{R} & \tilde{S} \\ 0 & 0 \end{pmatrix} \tag{6}$$

Now suppose that we want to fix one more entry, $\tilde{x_i}$, of the solution $\tilde{x}$. As demonstrated by equation (5) this corresponds to deleting a column from $\tilde{B}$ (or a row from $\tilde{B}^T$). Put $r = \text{rank}(\tilde{B})$. Let $\bar{B}^T$ be the matrix that is obtained from $\tilde{B}^T$ by removing row i. In Section 4.3.6 of [7] an efficient procedure is described for (partially) computing the QR decomposition of $\bar{B}^T$ from the QR decomposition of $\tilde{B}^T$. In this way we can avoid having to recompute the QR decomposition each time that we execute the Projection operation. The procedure makes extensive use of Givens rotations. We refer to Section 4.1.3 of

[7] for an introduction to Givens rotations. The procedure of updating a QR decomposition by Givens rotations is a relatively standard operation. In our case however, we want to update an *extended QR decomposition*, which involves some additional subtleties. The updating procedure from [7] leaves us with a decomposition

$$\bar{B}^T\bar{\Pi} = \bar{Q}\begin{pmatrix}\bar{R} & \bar{S}\\ 0 & 0\end{pmatrix}$$

but this might not be a valid extended QR decomposition. If the rank of $\bar{B}^T$ is smaller than the rank of $\tilde{B}^T$, the matrix $\bar{R}$ will be singular: it will contain a 0 on the main diagonal. In that case, $\text{rank}(\bar{B}^T) = \text{rank}(\tilde{B}^T) - 1$, and we have to apply a second procedure, as follows.
Let i be the smallest index such that $\bar{R}$ contains a 0 on the main diagonal in column i. We denote the nonzero part of this column by v. $\bar{R}_i$ is a linear combination of the columns $\bar{R}_1, \ldots, \bar{R}_{i-1}$. We will now construct a valid extended QR decomposition of $\bar{B}^T$. First, we apply a permutation to $\bar{B}^T$ that moves column i to the last position (shifting subsequent columns one position to the left):

$$\bar{B}^T\hat{\Pi} = \bar{Q}\begin{pmatrix}\hat{R} & \bar{S} & v\\ 0 & 0 & 0\end{pmatrix}$$

Because $\text{rank}(\bar{B}^T) = \text{rank}(\tilde{B}^T) - 1$, the columns of $\hat{R}$ are linearly independent. The matrix $\hat{R}$ is not guaranteed to be uppertriangular, because the band directly below the main diagonal may contain nonzero elements. This type of matrix is known as *Upper Hessenberg*. By performing a sequence of Givens rotations we can reduce the decomposition to the desired extended QR decomposition. This reduction can be found in Section 4.1.3 of [7].

4.3 Updating the projection: Releasing variables

Another updating operation that we will use for improving the performance of the Projection operation is just the opposite operation of fixing a solution variable: releasing a variable, such that its value is no longer fixed. In the same way as fixing a variable corresponds to removing a row from the QR decomposition of $\tilde{B}^T$, releasing this variable corresponds to reinserting this row in the decomposition. Fortunately, this is also an operation that can be performed efficiently by Givens rotations. Section 4.3.5 of [7] describes the procedure for a standard QR decomposition. We will outline the additional steps that extend the procedure to the case of extended QR decompositions.
Suppose that we have already computed the extended QR decomposition of $\tilde{B}^T$, as in equation (6). Let w be the row that has to be reinserted. Put $r = \text{rank}(B^T)$. By applying a sequence of Givens rotations (see Section 4.3.5 of [7]) we can transform the decomposition of $\tilde{B}^T$ into a decomposition of the form

$$\begin{pmatrix}\tilde{B}^T\\ w\end{pmatrix}\bar{\Pi} = \bar{Q}\bar{T} = \bar{Q}\begin{pmatrix}\bar{R} & \bar{S}\\ 0 & 0\\ 0 & \bar{w}_2\end{pmatrix}.$$

where $\bar{R}$ is uppertriangular and nonsingular, and the first r entries of the last row of $\bar{T}$ are zero. To transform this decomposition into an extended QR decomposition, we have to perform a few extra steps. First move the row containing $\bar{w_2}$ up, to just below $\bar{R}$ and $\bar{S}$, by applying a permutation to the rows of $\bar{T}$. By applying the same permutation to the columns of $\bar{Q}$, the decomposition remains valid. If all entries of $\bar{w_2}$ are zero, we are done. Otherwise, apply a permutation to the columns of $\bar{T}$, that swaps the column corresponding to the first nonzero entry of $\bar{w_2}$ with column $r+1$. This will result in a valid extended QR decomposition. In this case, appending w has increased the rank of $\bar{B^T}$.

4.4 An improved Projection operation

In order to make the Projection operation more efficient, we try to avoid recomputing the QR decomposition as much as possible, by using the update procedures. In addition, we attempt to reduce the number of update operations. We will now describe the method that we use to achieve these goals.
The global structure of the Projection operation is shown in Figure 1. In every

```
i := 0;
repeat
   i := i + 1;
   F_i := ∅;
   for j := 1 to mn do
   begin
      if (|x_j − 0.5| > p) then F_i := F_i ∪ {j};
      fix the values of all variables in F_i;
      compute the projection of the origin;
   end;
until (stop criterium has been met);
```

Figure 1: Global structure of the Projection operation

iteration of the outer loop, the value of all entries in F_i becomes fixed and the projection of the origin onto the corresponding linear manifold is computed. The outer loop is repeated until either the linear system has no solutions or all entries are close enough to the interval $[0, 1]$. Note that all variables that were fixed during the Projection operation can be modified again after the Projection operation has completed, by applying mills.
Our experimental results indicated that when the projection operation is executed a second time, the set F_i of variables that are fixed in iteration i of the outer loop is usually not very different from the set F'_i of fixed variables in the corresponding iteration of the previous Projection operation.
We use this information in the following way. Instead of just one QR decomposition, we store a table of QR decompositions and their corresponding sets of fixed entries. In every iteration of the outer loop, the QR decomposition that

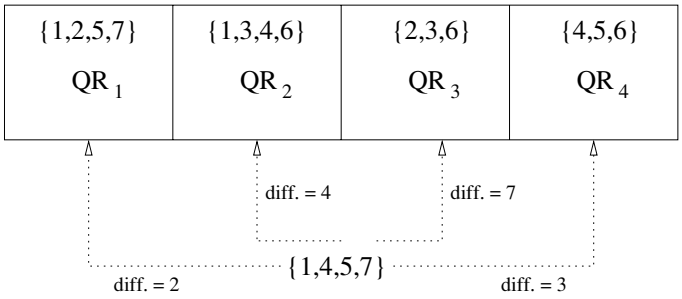

Figure 2: The new set of fixed variables, $\{1, 4, 5, 7\}$, is compared to all sets in the table.

corresponds to the current set of fixed variables is stored in the table. When a new projection of the origin has to be computed, at the end of the main loop of the Projection operation, the table is checked first to see if there is any QR decomposition available for which the set of fixed variables is almost the same as the current set (i.e., the cardinality of the symmetric difference between both sets is small). If this is the case, we can transform this stored QR decomposition into the required decomposition by applying the update operations from Section 4.2 and 4.3. Figure 2 shows how the table is searched for a suitable stored decomposition.
If the number of required update steps is too large, the QR decomposition is recomputed entirely. Let v be the cardinality of the minimal symmetric difference between the current set of fixed entries and all sets in the table. Let k denote the number of free (nonfixed) entries left. Then the algorithm will choose to recompute the QR decomposition if $k < \alpha v$, where $\alpha \in \mathbb{R}$ is a constant.
After the new decomposition has been computed, it is inserted into the table. To avoid that the table uses too much memory space, it has a maximum size, K. When the table is full and a new decomposition must be inserted, another decomposition is removed. The choice which decomposition will be removed is made according to timestamps: as a decomposition is inserted in the table it receives a timestamp. Each time that this decomposition is used to compute a new decomposition, it remains in the table and its timestamp is refreshed. When the table is full, the decomposition having the oldest timestamp is removed. All other decompositions in the table have been used more recently, so they remain in the table.

5 Experimental results

We implemented the algorithm from [6] in C++, using the gcc compiler. In comparison to the MATLAB-implementation of which the results are described in the original paper, the current implementation has been optimized in several ways.

The mills m_{uv} are represented as a list of entries instead of a vector or matrix. Because the mills are very sparse, the new representation is much more efficient. We implemented specially tuned matrix- and vector-libraries, which are very efficient in terms of both memory usage and execution speed.
In order to speed up the operation of finding an entry x_i for which a certain function $f(x_i)$ is maximal, priority queues implemented as binary heaps are used. This operation has to be performed at several points in the algorithm. We remark that this operation is not deterministic since multiple entries can have the same function value. This property allows the algorithm to be repeated multiple times (making different choices) with different results. For one test case, in Section 5.2, we used this property by performing more than one run, to get the best result.
For the implementation of the basic linear algebra routines, such as the extended QR decomposition, we use the linear algebra package LAPACK [1].
We have tested the algorithm with various types and sizes of matrices, using two different versions of the algorithm:
(**A**) Without QR updating. The QR decomposition of the matrix $\tilde{B}$, defined in equation (5), is computed from scratch in each iteration of the main loop of the Projection operation.
(**B**) With QR updating, as described in Section 4.4. For all tests, we used $\alpha = 10$ and $K = 8$ (see Section 4.4).
In order to allow for a good comparison between the results presented here and the results presented in [6], we have used similar test data. We used an Athlon 700MHz PC for all tests. We remark that this PC is faster than the Celeron 566 MHz PC used in the original paper, so the comparison of run times is not completely fair. For the two-dimensional test cases all linesums in the horizontal, vertical, diagonal and antidiagonal directions are given.
Very small floating point errors sometimes caused differences between runs of version (A) and (B) on the same test data.

5.1 Random images

The first set of test cases consists of random binary matrices of various sizes and densities. By *density* we mean the relative number of 1's. For the sizes 25×25 and 35×35 and for each of the densities 0.05, 0.10 and 0.5 of 1's, we have performed ten test runs. The results are summarized in Table 1, 2 and 3. These tables contain two characteristics: *#binary output* (the number of cases where the outcome is a binary matrix) and *average runtime* (in seconds).

The algorithm is not able to find a binary solution for many of the low-density test cases. For most test cases it finds either a binary solution or an integer solution that has only a small number (< 15) of nonbinary entries and those nonbinary entries have small absolute values. For some of the test cases however the algorithm finds integer solutions with many nonbinary entries, some of which have high absolute values. In all cases, version (B) (with projection updating) clearly outperforms version (A). In particular, version (B) is much faster for the

problem size	25 × 25			35 × 35	
algorithm	old	A	B	A	B
#binary output (of 10)	7	7	7	0	0
average run time (s)	1312	22	18	193	161

Table 1: Results for random test cases with density 0.05

problem size	25 × 25			35 × 35	
algorithm	old	A	B	A	B
#binary output (of 10)	4	2	3	4	3
average run time (s)	10661	35	22	1178	352

Table 2: Results for random test cases with density 0.10

problem size	25 × 25			35 × 35	
algorithm	old	A	B	A	B
#binary output (of 10)	10	10	9	10	10
average run time (s)	12350	136	18	1670	121

Table 3: Results for random test cases with density 0.50

examples of density 0.50. For that type of problem instances, the main loop of the Projection operation is executed many times, every time only fixing a small number of new variables, making efficient updates of the QR decomposition possible.

5.2 Structured images

In the original paper, the results for several test cases that originate from [3] are presented. According to the authors of that paper, these cases represent crystal structures. To allow for a good comparison between the implementation from [6] and the current implementation we have used two of these test cases, denoted by T_1 and T_2. We added a third example, T_3, which consists of two large areas of 1's such that there are very few lines that are constant. We will refer to the examples T_1, T_2 and T_3 by the term *structured examples*. The test cases T_1 and T_2 correspond to the functions f_4 and f_6 respectively from [6]. The three test matrices are shown in Figure 3 (1's are represented by black squares, 0's by white squares). The results for the structured examples are summarized in Table 4. Besides the run time the table shows the number of nonbinary entries in the result. The results from [6] are labeled *old*.

problem	T_1			T_2			T_3	
problem size	29×46			36×42			40×40	
algorithm	old	A	B	old	A	B	A	B
run time (s)	463	28	28	2901	58	55	238	188
#nonbinary entries	0	0	0	0	0	0	6	2

Table 4: Results for the structured examples

For the cases T_1 and T_2 each of the new implementations was able to recon-

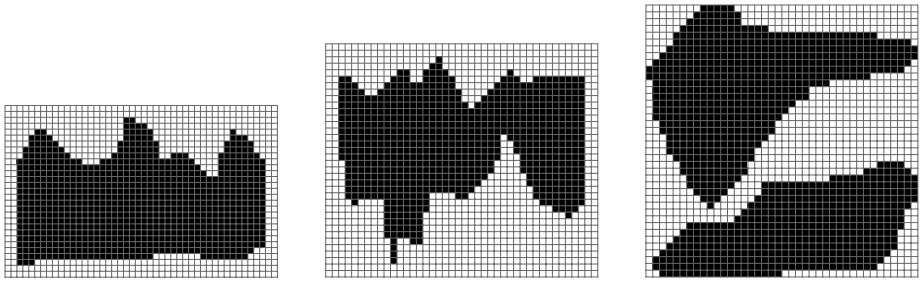

Figure 3: Three structured test images.

struct the original matrices exactly. The case T_3 turned out to be much harder. We had to repeat the algorithm several times, with different random seeds) to obtain reasonable results. For T_3 the best result (least nonbinary entries) of 5 runs with different random seeds are shown.
For this type of problem instances, projection updating results in less improvements of the run time than for the random instances. This is due to the fact that for these instances the Projection operation is executed infrequently and every time a large number of new variables is fixed. As a result, recomputing the QR decomposition is often more efficient than updating a decomposition from the table.

5.3 Three-dimensional example

All functions $A \to \mathbb{R}$ in the current implementation are stored in the vector representation, as opposed to the matrix representation. This allows the algorithm to handle three-dimensional instances just as it can handle the two-dimensional case. We have performed an experiment with a three-dimensional instance. The only modification to the program concerns the input- and output-routines and the choice of the set of mills. We studied the case where the projections in the three directions parallel to the axes are given. In this case the collection of mills

consists of the translates of the block

$$\begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

where the second 2×2-matrix is stacked on top of the first one.
We tested the algorithm with a problem which has the property that very few lines are constant. Because the purpose of this test is to show if the algorithm is capable of solving three-dimensional instances we only used implementation A. The test problem, a $10 \times 10 \times 10$ image, is shown in Figure 4 as a set of two-dimensional slices. The algorithm found a binary solution in 1600 seconds. This solution is quite different from the original set of matrices. Apparently using this set of directions in three dimensions leaves much freedom in the solution space.

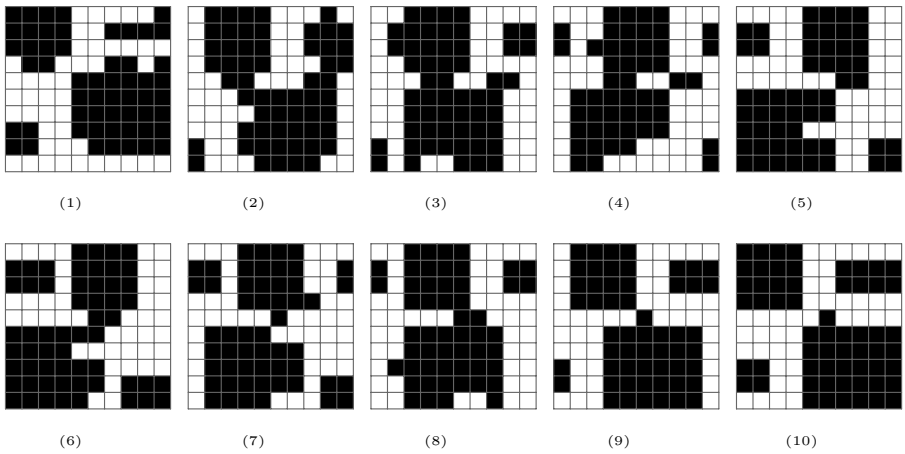

Figure 4: Slices of the three-dimensional test image.

6 Conclusions

The current implementation of the algorithm from [6] is much faster than the implementation used in the original paper regardless of whether projection updating is used or not. The new implementation has the added advantage that it can be used for higher-dimensional arrays without any significant modification. We have described how projection updating can lead to a reduction in the time complexity of the Projection operation in some cases. For all of the random test cases, of densities 0.05, 0.10 and 0.50, projection updating leads to a significant decrease in run time in comparison to recomputing the projection from scratch. For the structured examples, projection updating results in a small improvement of the run time, because the Projection operation is executed infrequently and every time a large number of new variables is fixed.

The structured example T_3 and the random test cases of low density show that for certain test cases the algorithm is not able to find a binary solution. Improving the quality of the resulting reconstruction will be the subject of further research.
The test results indicate that the algorithm is capable of solving moderately large instances. We consider our new version of the algorithm to be a promising approach to solving the reconstruction problem, and believe that there is sufficient potential for even further improvements.

7 Acknowledgments

The author would like to thank Herman te Riele and Robert Tijdeman for their valuable comments and suggestions.

References

[1] E. Anderson, editor, *LAPACK: A portable linear algebra library for high-performance computers*, SIAM, Philadelphia, 1992.

[2] R.E. Cline and R.J. Plemmons, ℓ_2-solutions to Underdetermined Linear Systems, *SIAM Review.* **18**, 1976, 92–106.

[3] P. Fishburn, P. Schwander, L. Schepp and R.J. Vanderbei, The Discrete Radon Transform and its Approximate Inversion via Linear Programming, *Discr. Appl. Math.*, **75**, 1997, 39–61.

[4] G.H. Golub and C.F. Van Loan, *Matrix Computations, 3rd edition*, Johns Hopkins University Press, Baltimore, MD, 1996.

[5] L. Hajdu and R. Tijdeman, Algebraic aspects of discrete tomography, *J. Reine Angew. Math.* **534**, 2001, 119–128.

[6] L. Hajdu and R. Tijdeman, An algorithm for discrete tomography, *Lin. Alg. Appl.* **339**, 2001, 147–169.

[7] G.W. Stewart, *Matrix Algorithms I: Basic Decompositions*, SIAM, Philadelphia, 1998.

[8] K.J. Batenburg, Analysis and optimization of an algorithm for discrete tomography. In: A. Del Lungo, V. Di Gesù and A. Kuba, editors: *Electronic Notes in Discrete Mathematics*, volume 12, Elsevier, 2003.